Title	UPS Serial Programming Guide
No.	

Rev.	Date	By	History
0.0.1	2003-04-09	UIS	Initial draft
0.0.8	2003-05-21	KCY	Title of this document, Chapter 2.1
0.0.9	2003-07-09	UIS	Easytrap With UPSLink100 2.2.1 upgrade

Index

1. Overview....3
    1.1. Basic Structure....3
    1.2. Components....4
2. User Serial Programming....6
    2.1. Preparation....6
    2.2. ups_mon.c....7
    2.3. UPS data....8
    2.4. The Library : libupslink.a....8
    2.5. Modifying ups_app.c....15
    2.6. Modifying the Makefile and Compiling....23
3. Download and Execution....24
    3.1. Download....24
    3.2. Running the UPS Serial Program....24
4. Add User Defined Information....25
    4.1. User Defined UPS Alarm....25
    4.2. User Defined UPS Information....26
    4.3. User Defined UPS Status Information....27
    4.4. User Defined UPS Configuration....28
    4.5. User Defined UPS Control....31
    4.6. User Defined UPS Test....33

1. Overview

UPSLink is set to monitor and control a UPS based on the UPS serial protocol given by Sena Technologies. In addition to this protocol, a user defined UPS serial protocol could be also implemented in UPSLink. A user can write a serial program which implements a UPS specific serial protocol at minimum efforts.

Sena Technologies supports the user customization by providing source codes, library files, Sena UPS serial protocol and the serial program written on the Sena protocol.

As a SNMP agent, UPSLink completely complies with RFC1628(UPS MIB, Information base for UPS administration on the network) and exports various functions to administer a UPS via Network. Just by modifying UPS serial program to reflect a particular protocol, a user can attach UPSLink to the user's UPS.

In addition, user defined alarms (including SNMP trap), tests, monitoring, configuration, control, Web page contents could be changed.

1.1. Basic Structure

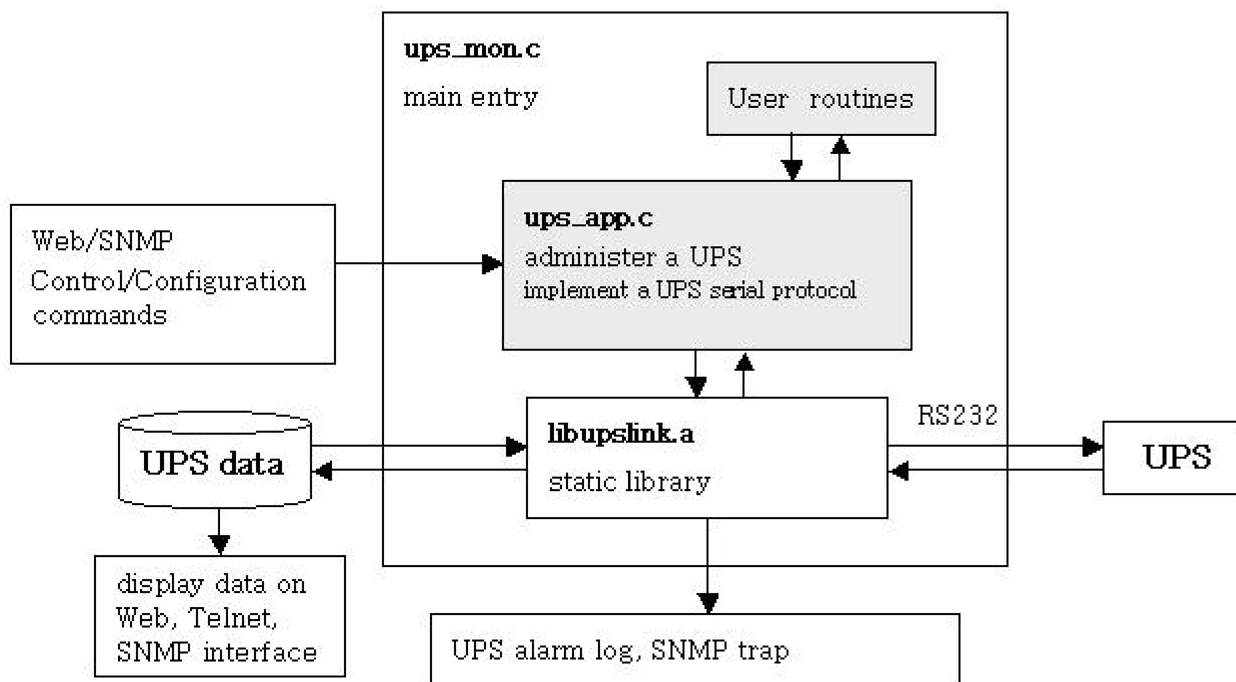

Figure 1-1 UPS Serial Program Diagram

The shaded boxes in the figure 1-1 should be replaced or added by a user. Other parts in this diagram are provided as source codes or library.

1.2. Components

The only file that a user should touch is ups_app.c. Other files are references for users to write their own codes.

ups_mon.c

This file contains basic functions as of a main entry to the UPS serial program, program initialization, program termination, data initialization, opening/closing of a serial port and so on.

Note : A user doesn't need to modify this file.

ups_data.h

UPS data structures are defined here. They include the RFC1628 defined items and user defined items.

Note : A user cannot modify this file.

ups_alarm.h

UPS alarms are listed here.

libupslink.a

This library provides an communication agent between UPSLink and UPS via serial interface. UPS data parsed in ups_app.c are stored and accessed by this library. Logging alarm, SNMP trap and emailing are supported by this library. These all actions are automated or called from ups_app.c. The functions exported from this library are defined in several header files in the 'include' directory.

Note : This file is a static library that a user program links when it's built.

libups_data.h, ups_msgq.h, ups_ser.h, ups_sem.h, strqueue.h and ups_easytraplog_msgq.h

The function prototypes of libupslink.a are defined here.

The libups_data.h covers UPS data management, alarm sending and etc. The ups_msgq.h is for Web and SNMP. The ups_ser.h and strqueue.h are for serial communication and serial data buffer. The ups_easytraplog_msgq.h helps to send a Non-RFC1628, user-specific SNMP v1 trap.

Note : The string queue related functions in strqueue.h are useful in manipulating ASCII characters. If data is dealt in binary format, a user should directly change the data read from ups_ser_read() instead of utilizing the string queue related functions.

Note : A user may not modify this file.

ups_app.c

This implements UPS serial protocol, parses the data from a UPS serial port, periodically monitors a UPS and handles asynchronous UPS messages. A user could modify from the template codes or write a code in a separate file and call it from this file.

Separate process is created here to handles user commands from SNMP or Web. Multiple processes access a serial port simultaneously in a thread-safe way.

Note : A user should modify this file.

2. User Serial Programming

2.1. Preparation

Prepare the development environment as follows.

1) Get a PC that is capable of CD-ROM booting. It has to have 2GB+ hard disk space.

2) Install Redhat 7.0

- ?? Download Redhat 7.0 from Redhat web site
- ?? Burn CDs with the image
- ?? Configure BIOS to boot from CD-ROM and boot the PC with Redhat 7.0 CD in the CD-ROM.
- ?? Install Redhat.

3) Install Hardhat (Refer `docs/HHL-JE-20.pdf` in Hardhat CD)

- ?? After logging in as root into Redhat, proceed with Hardhat installation by inserting Hardhat CD in the CD-ROM and typing as below.

```
mount /mnt/cdrom      [enter]
/mnt/cdrom/bin/hhl-host-isntall     [enter]
```

- ?? If asked which LSP to install, `embeddedplanet-rpxlite` is the choice.
- ?? Hardhat Linux is installed under `/opt/hardhat`.
- ?? Export the path.

```
export  PATH=$PATH:/opt/hardhat/devkit/ppc/8xx/bin:/opt/hardhat/host/bin
[enter]
```

4) Compilation of the template source

- ?? Copy `UPS_serial_program_template.tar.gz` from the UPSLink CD to a working directory.
- ?? Decompress the file.

```
tar –xzvf upslink_app_template.tar.gz [enter]
```

- ?? Verify the contents.

```
#cd upslink_template [enter]
#ls [enter]
Makefile    include    lib    ups_app.c    ups_mon.c
#ls include [enter]
libups_data.h    strqueue.h    ups_alarm.h    ups_data.h    ups_msgq.h
ups_sem.h        ups_ser.h
#ls lib [enter]
libupslink.a
```

- ?? Compile as it is.

```
#make [enter]
```

- ?? Verify the results.

```
#ls [enter]
Makefile    include    lib    ups_app.c    ups_app.o    ups_mon.c
ups_mon.o   upsapp
```

The `upsapp` here is a template application. A user should modify `ups_app.c` to implement a specific

UPS serial protocol or add additional features. See the next chapter.

2.2. ups_mon.c

This file is not necessarily modified. In most cases, this doesn't need a change.

The size of a string buffer for the serial data is defined and the buffer queue for the string buffer is declared here. A user may change the size or eliminate the string buffer if serial data are binary not ASCII.

```
#define QUEUE_SIZE 2048
queue upsq;
```

This initializes UPS serial program. This registers a signal handler for a process and initializes UPS data, message queue, semaphore, string buffer queue and etc. Binary serial protocol needs not string buffer queue initialization.

```
void init_ups_mon(bool bfirst){
	signal(SIGINT, ups_mon_sighandler);
	signal(SIGTERM, ups_mon_sighandler);
	signal(SIGCHLD, ups_mon_sighandler);

	init_upsdata(bfirst);
	init_upstraplogmsgq();
	init_upseasytraplogmsgq();
	init_upssem();
	init_upsmsgq();

	// init data queue
	if(queue_init(&upsq, QUEUE_SIZE) < 0)	{
		fprintf(stderr, "queue init failed.. exiting..\n");
		exit(0);
	}
	// open serial device
	if(ups_ser_open()<0)  {
		fprintf(stderr, "Can't open serial device.. exiting..\n");
		exit(0);
	}
}
```

This is a process signal handler. The signals that are delivered at the process termination are handled here.

```
void ups_mon_sighandler(int sig){
	switch(sig){
		case SIGINT:
		case SIGTERM:
			ups_ser_close();
			finalize_upsdata();
			queue_free(&upsq);
			exit(0);
		case SIGCHLD:
			{
				int status;
				pid_t pid;

				pid = waitpid(-1, &status, WNOHANG);
```

```
                    if (pid <= 0)
                          break;
               }
               break;
     }
}
```

The main function calls `start_ups_app()` in `ups_app.c`.

```
main(){
     printf("Starting ups_mon.. \r\n");
     if(!fork()){
          init_ups_mon(true);
          start_ups_app();
     }
}
```

2.3. UPS data

ups_data.h

UPS data structure is loaded on the memory of UPSLink. Its format is declared here. The data format encloses all the RFC1628 items and user defined items.

Direct access to these data is not allowed. Instead, a user calls the functions defined in `libups_data.h`. The UPS data are composed of a number of sub-structures, which are ID, battery, input, output, bypass, alarm, configuration, control, test, others and user area. The UPS data are read or written to the memory on the basis of these units.

ups_alarm.h

UPS alarms are enumerated here. A user needs to consult this file when alarm related jobs are required.

2.4. The Library : libupslink.a

The library, `libupsink.a,` is provided as compiled. Here explains the included functions.

libups_data.h

`libups_data.h` defines UPS data functions.

`init_upsdata` : initializes UPS data. This is called at the program startup in `init_ups_mon.` A user doesn't need to call this.

```
void init_upsdata(bool bfirst);
```

`reset_upsdata` : initializes some of the UPS data; ID, battery, input, output and bypass group. A user calls this for the cases like UPS communication lost alarm.

```
void reset_upsdata();
```

`finalize_upsdata` : finalizes UPS data. This is called at the program termination. See `ups_mon_sighandler`. A user doesn't need to call this.

```
void finalize_upsdata();
```

Alarm group

`register_usralarm` : appends a user defined alarm. See chapter 4 for details.

```
int register_usralarm(int usrindex, int oiddepth, unsigned long *oid,
char *shortdesc, char *longdesc);
```

`read_usralarm` : reads a user defined alarm of which index is the `usrindex`.

```
int read_usralarm(int usrindex, struct _usr_alarm_object *alarmobj);
```

`set_alarm` : Call this function when an alarm is set or reset. If the third argument, `'force'`, is set to `'false'`, the alarm status is examined to see if changed. Only on the status change, time is recorded and SNMP trap or email is sent as configured. If the `'force'` is set to `'true'` all the alarm actions are taken regardless of the status change. The `'alarmindex'` value is referenced in `ups_alarm.h`. The `'status'` takes `ALARM_RELEASED` or `ALARM_PRESENT`.

```
int set_alarm(int alarmindex, ALARM_STATUS status, bool force);
```

RFC1628 rule: The alarms gone off by set_alarm(..) get unique upsAlarmId. If upsAlarmTestInProgress is released, upsTrapTestCompleted is sent to a trap receiver server and upsTestSpinLock is incremented so as to change the UPSLink mode to be ready to test on SNMP-set command. The upsTrapAlarmEntryAdded trap is not sent for upsAlarmTestInProgress and upsAlarmOnBattery. While upsAlarmOnBattery is set, upsTrapOnBattery trap is set every one minute.

Example :

```
set_alarm(_ALARM_COMMLOST, ALARM_PRESENT, false);
```

`get_alarmstatus` : reads alarm status. The return values are either `ALARM_RELEASED` or `ALARM_PRESENT`. The `'alarmindex'` is referenced in `ups_alarm.h`.

```
int get_alarmstatus(int alarmidex);
```

`get_present_alarmnum` : returns the number of alarms that are currently set.

```
int get_present_alarmnum();
```

Information and status group

`register_usrinfo, read_ups_usrinfo, save_ups_usrinfo` : These functions are related to user defined UPS information. See Chapter 4.

```
int register_usrinfo(int usrindex, char *desc, INFO_TYPE type);
int read_ups_usrinfo(int usrindex, int *infoval, char *infostr);
int save_ups_usrinfo(int usrindex, int infoval, char *infostr);
```

`register_usrstatus, read_ups_usrstatus, save_ups_usrstatus` : These functions are related to user defined UPS status. See Chapter 4.

```
int register_usrstatus(int usrindex, char *desc, STATUS_TYPE type);
int read_ups_usrstatus(int usrindex, int *statusval, char *statusstr);
int save_ups_usrstatus(int usrindex, int statusval, char *statusstr);
```

read_ups_identification, save_ups_identification : reads and writes UPS ID information to the UPS data area.

_UPS_DATA_ID is defined in ups_data.h.

```
int read_ups_identification(_UPS_DATA_ID *data);
int save_ups_identification(_UPS_DATA_ID *data);
```

Note : The 'agent_sw_ver' member of _UPS_DATA_ID is write-protected.

Example :

```
_UPS_DATA_ID data;
read_ups_identification(&data);
strncpy(data.manufacturer, "My Manufacturer", sizeof(data.manufacturer)-1);
save_ups_identification(&data);
```

read_ups_battery, save_ups_battery : reads/writes UPS battery information from/onto the UPS data area.

_UPS_DATA_BATTERY is defined in ups_data.h.

```
int read_ups_battery(_UPS_DATA_BATTERY *data);
int save_ups_battery(_UPS_DATA_BATTERY *data);
```

Note : Please give notice to physical unit of each member of _UPS_DATA_BATTERY structure. For example, voltage and current have 0.1[VDC] and 0.1[amp DC] as their units respectively. The ups_data.h explains more about this.

read_ups_input, save_ups_input : reads/writes UPS input information from/onto the UPS data area. _UPS_DATA_INPUT is defined in ups_data.h.

```
int read_ups_input(_UPS_DATA_INPUT *data);
int save_ups_input(_UPS_DATA_INPUT *data);
```

Note : Please give notice to physical unit of each member of _UPS_DATA_INPUT structure. For example frequency and current have 0.1[Hz] and 0.1[RMS amp] as their units respectively. See the ups_data.h.

read_ups_output, save_ups_output : reads/writes UPS output information from/onto the UPS data area. _UPS_DATA_OUTPUT is defined in ups_data.h.

```
int read_ups_output(_UPS_DATA_OUTPUT *data);
int save_ups_output(_UPS_DATA_OUTPUT *data);
```

Note : Please give notice to physical unit of each member of _UPS_DATA_OUTPUT. For example, frequency and current have 0.1[Hz] and 0.1[RMS amp] as their units respectively. See the ups_data.h.

read_ups_bypass, save_ups_bypass : reads/writes UPS output information from/onto the UPS data area. _UPS_DATA_BYPASS is defined in ups_data.h.

```
int read_ups_bypass(_UPS_DATA_BYPASS *data);
int save_ups_bypass(_UPS_DATA_BYPASS *data);
```

Note : Please give notice to physical unit of each member of _UPS_DATA_BYPASS. For example, current has 0.1[RMS amp] as its unit. See the ups_data.h.

Control group

read_ups_control_shutdowntype, save_ups_control_shutdowntype : reads/writes

UPS shutdown type from the UPS data area.

```
int read_ups_control_shutdowntype(SHUTDOWN_TYPE *type);
int save_ups_control_shutdowntype(SHUTDOWN_TYPE type);
```

`read_ups_control_autorestart, save_ups_control_autorestart` : reads/writes UPS auto-restart type from the UPS data area.

```
int read_ups_control_autorestart(AUTO_RESTART *type);
int save_ups_control_autorestart(AUTO_RESTART type);
```

`save_ups_control_shutdownafterdelay` : This function should be called along with sending scheduled-shutdown command to a UPS. By doing this, the time left to the UPS shutdown is correctly tracked on the Web page.

```
int save_ups_control_shutdownafterdelay(long delay);
```

`save_ups_control_startupafterdelay` : This function should be called along with sending scheduled-startup command to a UPS. By doing this, the time left to the UPS startup is correctly displayed on the Web page.

```
int save_ups_control_startupafterdelay (long delay);
```

`save_ups_control_rebootwithduration` : This function should be called along with sending scheduled-restart command to a UPS. By doing this, the time left to the UPS restart is correctly displayed on the Web page.

```
int save_ups_control_rebootwithduration(long duration);
```

`register_usrcontrol` : registers user defined UPS control item. See chapter 4.

```
int register_usrcontrol(int usrindex, char *desc, CONT_TYPE type, int defval, char *defstr, char **combo_desc);
```

Configuration group

`read_ups_config, save_ups_config` : reads/writes UPS configuration information from/onto UPS data area. `_UPS_DATA_CONFIG` is defined in `ups_data.h`.

```
int read_ups_config(_UPS_DATA_CONFIG *data);
int save_ups_config(_UPS_DATA_CONFIG *data);
```

Note : Please give notice to physical unit of each member of `_UPS_DATA_CONFIG`. For example, frequency and voltage have 0.1[Hz] and 1[RMS volts] as their units respectively. The `ups_data.h` explains about this.

`register_usrconfig, read_ups_usrconfig, save_ups_usrconfig` : These are related with user defined UPS configuration. See chapter 4.

```
int register_usrconfig(int usrindex, char *desc, CONT_TYPE type, char **combo_desc);
int read_ups_usrconfig(int usrindex, int *configval, char *configstr);
int save_ups_usrconfig(int usrindex, int configval, char *configstr);
```

Test group

`register_usrtest, read_usrtest` : These are related with user defined UPS test. See

chapter 4.

```
int register_usrtest(int usrindex, int oiddepth, unsigned long *oid,
char *testdesc);
int read_usrtest(int usrindex, struct _usr_test_object *testobj);
```

`read_ups_testid` : reads the last test ID which corresponds to `upsTestId` in RFC1628. If the return value is not `TEST_USER_SPECIFIC`, `upsTestId` is `.1.3.6.1.2.1.33.1.7.7.RETURN_VALUE` and `ups_testindex` is neglected. Otherwise, `upsTestId` is `usrtestobject.oid` returned by `read_usrtest(usr_testindex, usrtestobject)`.

```
UPS_TEST read_ups_testid(int *usr_testindex);
```

`save_ups_testid` : saves the last test ID which corresponds to `upstTestId` in RFC1628. RFC1628 designates that `upsTestId` is to be saved by calling this function on each UPS test.

```
int save_ups_testid(UPS_TEST testid, int usr_testindex);
```

RFC1628 rule : By calling this function the `start_time` member of `_UPS_DATA_TEST` is updated and the `elapse_time` member of `_UPS_DATA_TEST` is correctly calculated at the completion of a test. If it's a user defined test pass `TEST_USER_SPECIFIC` for the '`testid`' argument and correct value for the `usr_testindex`. If the test is not user defined, pass 0 for the `usr_testindex` argument. If the `testid` argument is `TEST_ABORT`, the `elapse_time` member of `_UPS_DATA_TEST` is recalculated and the '`summary`' member of `_UPS_DATA_TEST` is set to `TESTRES_ABORTED`. The '`summary`' is set to `TESTRES_INPROG` in other cases.

`read_ups_test_summary`, `save_ups_test_summary` : reads/writes UPS test results from/onto the UPS data area.

```
UPS_TESTRES read_ups_test_summary();
int save_ups_test_summary(UPS_TESTRES summary);
```

`read_ups_test_detail`, `save_ups_test_detail` : reads/writes detailed UPS test results from/onto the UPS test area.

```
int read_ups_test_detail(char *detail);
int save_ups_test_detail(char *detail);
```

`get_ups_test_starttime` : returns the elapsed time between UPSLink startup and last test in the measure of one hundredth seconds. If no test was done after UPSLink startup, 0 is returned.

```
unsigned long get_ups_test_starttime();
```

`get_ups_test_elapsetime` : returns the elapsed time of the last test of an ongoing test in the measure of one hundredth seconds. 0 if returned if no test has been executed.

```
unsigned long get_ups_test_elapsetime();
```

ups_ser.h

`ups_ser.h` declares RS232 (communication between UPSLink and a UPS) related functions.

`ups_ser_open` : opens the serial port. This is called at the program startup in `ups_mon.c`. A

user needs not call this function.

```
int ups_ser_open();
```

`ups_ser_close` : closes the serial port. This is called at the program exit in `ups_mon.c`. A user needs not call this functin.

```
void ups_ser_close();
```

`ups_ser_write` : writes the 'len' bytes of the 'buff' to the serial port. The number of bytes written is returned.

```
int ups_ser_write(const char *buff, int len);
```

`ups_ser_read` : reads the 'len' bytes of data from the serial port and saves them to the 'buff'. The actually number of bytes read is returned.

```
int ups_ser_read(char *buff, int len);
```

ups_msgq.h

This header file declares the Web based configuration/control/test commands or SNMP-set commands that are transferred to the UPS serial program. These commands are enumerated.

`init_upsmsgq` : initializes UPS message queue. This is called at the program startup. A user needs not call this.

```
void init_upsmsgq(void);
```

`send_upsmsgq` : sends UPS commands into the message queue. This is called by the Web server or SNMP daemon. A user never calls this function.

```
int send_upsmsgq(int command, long value, char *str);
```

`read_upsmsgq` : retrieves messages stacked in the UPS message queue. The `ups_app.c` template code has this function call. A user needs to modify `handle_upsmsg(..)` to take actions to the retrieved messages. UPS commands are enumerated in `ups_msgq.h`. The 'value' and 'str' arguments are auxiliary data to the 'command'. Refer to "Modifying ups_app.c" for the actual usage of this function.

```
int read_upsmsgq(int *command, long *value, char *str);
```

Note : This function is a blocking one.

ups_sem.h

This is semaphore utility to provide thread-safe serial port access.

`init_upssem` : initializes semaphore. This is called from `ups_mon.c` at the program start up. A user needs not call this function.

```
int init_upssem();
```

`lock_upssem` : Locks the semaphore. Call this function before accessing the serial port. Unlock the semaphore after using the serial port by calling `unlock_upssem`.

```
int lock_upssem();
```

`unlock_upssem` : Unlocks the semaphore. Use this function in pair with `lock_upssem`.

```
int unlock_upssem();
```

strqueue.h

The serial data are stored in a string buffer. And a collection of utilities defined in this header file handle the buffer. If the serial protocol is written in binary data, these utilities are not applied. A global variable of the queue structure defined in `strqueue.h` is declared in `ups_mon.c`. A user can eliminate the declaration if binary protocol is used. Multiple buffers could be utilized on a user's demand.

```
typedef struct {
       char *buf;
       int maxsize;
} queue;
```

`queue_init` : allocate memory for the string buffer and initializes it. This is called at the program startup in `ups_mon.c`. A user needs not call this function. A binary UPS serial protocol doesn't need this function. Maximum size of the buffer is 65535 bytes.

```
int queue_init(queue *q, int size);
```

`queue_free` : releases the memory for the serial buffer. This function is called at the program exit. A user needs not call this function for a binary UPS serial protocol.

```
void queue_free(queue *q);
```

`queue_maxsize` : returns the size of the string buffer in bytes.

```
int queue_maxsize(queue *q);
```

`queue_size` : returns the size of the data in the string buffer in bytes.

```
int queue_size(queue *q);
```

`queue_write` : appends the `'len'` bytes of the `'buf'` to the string buffer `'q'`.

```
int queue_write(queue *q, char *buf, int len);
```

`queue_copy` : copies the `'buf'` to the `'q'`. The `'len'` bytes of the `'buf'` is copied to `'q->buf+start'`.

```
int queue_copy(queue *q, char *buf, int start, int len);
```

`queue_read` : cuts the `'len'` bytes from `'q->buf_start'` and pastes them to the `'buf'`. The string left from the cut is shifted to the front to fill the gap.

```
int queue_read(queue *q, char *buf, int start, int len);
```

queue_delete : deletes the string from 'q->buf+start' by 'len' bytes. The string in the back of the deleted portion is moved to the front by 'len' bytes.

```
int queue_delete(queue *q, int start, int len);
```

queue_clear : clears the string buffer.

```
int queue_clear(queue *q);
```

ups_easytraplog_msgq.h

As explained already in previous section, UPSLink generates RFC1628 UPS SNMP traps by utilizing set_alarm(..) function of libups_data.h. It is possible to generate NON-RFC1628 SNMP traps at user's convenience.

Note : The "trap-type" of trap receivers must be set to EASY_TRAP to be sent to this non-RFC1628 traps. This non-RFC1628 traps are SNMP v1 trap.

```
typedef enum {VB_INTEGER=1, VB_STRING=2, VB_OID=3} VB_TYPE;

struct _ups_easytrap_varbind{
       VB_TYPE type;
       unsigned long oid[MAX_OID_DEPTH];
       int     int_var;
       char    string_var[200];
       unsigned long oid_var[MAX_OID_DEPTH];
};

struct _ups_easytraplogmsg {
       unsigned long enterprise_oid[MAX_OID_DEPTH];
       int     generic;
       int     specific;
       struct _ups_easytrap_varbind varbind[10];
       int            sendtrap; // 0-do not send, 1-send
       int            sendmail; // 0-do not send, 1-send
       int            writelog; // 0-do not write, 1-write
       char    desc[100];
};
```

Construct a data structure referring to above definition and send the trap using send_upseasytraplogmsgq(struct _ups_easytraplogmsg msg). The number of variables that are to be bound into the trap is up to 10.

init_upseasytraplogmsgq : initializes easy trap message queue. This is called at the program startup. A user needs not call this.

```
void init_upseasytraplogmsgq(void);
```

send_upseasytraplogmsgq : generates easy trap.

```
int send_upseasytraplogmsgq(struct _ups_easytraplogmsg msg);
```

2.5. Modifying ups_app.c

The ups_app.c performs serial communication with a UPS, protocol parsing, data manipulation and delivery of user's configuration and control commands to a UPS. User defined items are added from here. See chapter 4 for details. Each function in the ups_app.c is explained here to help a user to understand its functionality.

Note : The ups_app.c is the basic structure that includes mandatory parts. A user does not need to construct the codes from the ground.

start_ups_app : This function is called from ups_mon.c's main function. Actual UPS serial program starts here. Two processes take a separate role. One monitors UPS and the other handles user's commands. Read the comments in the code snippet below.

```
void start_ups_app()
{
     pid_t pid; // process ID to be forked
     register_usrspec(); // register the user defined items
     queue_clear(&upsq); // clear the serial input buffer
                         // (only when ASCII protocol is used)
     pid = fork(); // spawn a child process
     if(pid == -1){ // exit if fails to spawn a new process
          fprintf(stderr, "process spawn fail. exiting..\r\n");
          exit(0);
     }
     else if(pid != 0){ // parent process, monitors a UPS
          // save this program's version,read and write on a structure basis
          _UPS_DATA_ID id;
          read_ups_identification(&id);
          strncpy(id.app_sw_ver, "user_app v1.0", sizeof(id.app_sw_ver)-1);
          save_ups_identification(&id);

          usleep(500000); // sleep to synchronize with other processes
                          // (0.5sec)

          // start monitoring a UPS
          while(1){{
                    // synchronize the access to the serial port,
                    //   thread-safe, lock the semaphore
                    lock_upssem();

                    // send commands/ read response/ handle asynch msg/...
                    // poll_device does this
                    poll_device();
                    // poll_device is explained later.

                    // unlock the semaphore
                    unlock_upssem();

                    // yield the CPU (1 sec)
                    usleep(1000000);
               }
          }
     }
     else{ // child process, handles user events
          while(1){
               // commands via Web or SNMP  are stored in the message queue
               int command;
               long val;
               char strval[256] = {0, };

               // read_upsmsgq : blocking function
```

```
                // pending until a message is received in the message queue
                read_upsmsgq(&command, &val, strval);

                // The message queue received messages.

                // lock the semaphore to synchronize
                //   the access to the serial port
                lock_upssem();

                // handle user’s commands
                // The ‘command’ argument values are listed in ups_msgq.h
                // The ‘val’ and ‘strval’ are auxiliary values
                //   to the ‘command’.
                handle_upsmsg(command, val, strval);

                // unlock
                unlock_upsssem();

                // sleep for a while to yield the CPU (1 sec)
                usleep(1000000);
          }
     }
}
```

`poll_device` : This function is called from the parent process’ while loop. A user can use this function to send a command to a UPS periodically. The `read_buffer` function will read and parse the response. This code example cycles the four commands to monitor a UPS.

```
enum { BATTERY=1, INPUT=2, OUTPUT=3, BYPASS=4};
int lastcmd = GET_BATTERY;

static int poll_device()
{
     char cmdstr[100] = {0, };

     switch(lastcmd)
     {
          case BATTERY:
               sprintf(cmdstr, “GET_BATTERY”);
               break;
          case INPUT:
               sprintf(cmdstr, “GET_INPUT”);
               break;

          case OUTPUT:
               sprintf(cmdstr, “GET_OUTPUT”);
               break;

          case BYPASS:
               sprintf(cmdstr, “GET_BYPASS”);
               break;
     }
     // send the string comman dto UPS
     send_command(cmdstr);
     usleep(500000); // wait for 0.5 sec
     read_buffer(lastcmd);

     if(lastcmd == BYPASS)
          lastcmd = BATTERY;
     else
          lastcmd ++;
}
```

`send_command` : sends a command to a UPS via the serial port. A user needs not change this function.

```
static int send_command(char *cmd, int len)
{
     int res;
     res = ups_ser_write(cmd, len);
     return res;
}
```

`read_buffer` : reads data from the UPS. After stores them in the string buffer queue this function interprets the serial data in the queue. (If binary UPS serial protocol is adopted, the queue is not applicable.)

The results of the interpretation are saved onto the UPS data. The last command is considered at this time. If an alarm is set or reset, `set_alarm` is used. If the read data are about asynchronous trap or alarm a user should handle this in a separate way.

Note : The last command is passed to the argument of this function. The command lists are defined by a user.

This code reads the serial port and save the data to the string queue. It searches for asynchronous traps and the response to the last command. It handles the search results. (ASCII protocol is assumed here.)

```
static int read_buffer(int lastcmd)
{
     int len;
     int totallen = 0;
     char buff[2048] = {0, };

     // read the serial port until it's empty. save it in the string queue.
     do{
          len = ups_ser_read(buff, sizeof(buff));
          if(len > 0){
               queue_write (&upsq, buff, len);
               totallen += len;
          }
     }while(len > 0);

     // try once more if nothing is read at the first try
     if(totallen == 0){
          usleep(500000); // wait 0.5 sec
          do{
               len = ups_ser_read(buff, sizeof(buff));
               if(len > 0) {
                    queue_write (&upsq, buff, len);
                    totallen += len;
               }
          }while(len > 0);

          // If no data is read at the double check,
          //   reset the UPS data and set an alarm for communication lost.
          if(totallen == 0){
               _UPS_DATA_ID id;
               reset_upsdata();
```

```
                // do not reset the version of this program
                read_ups_identification(&id);
                strcpy(id.app_sw_ver, "user_app v1.0");
                save_ups_identification(&id);
                set_alarm(_ALARM_COMMLOST, ALARM_PRESENT, false);
                return -1;
            }
    }
    // Release the "communication lost" alarm if a response was received
    set_alarm(_ALARM_COMMLOST, ALARM_RELEASED, false);

    // search the queue for an asynchronous trap and handle it if any
    catch_all_async_traps();

    // search the response for the last command and handle the response
    catch_response(lastcmd);

    // clear the string buffer queue
    queue_clear(&upsq);
    return 0;
}
```

`catch_all_async_traps` : catches asynchronous traps from the string queue.

```
static int catch_all_async_traps()
{
    // search the string queue, upsq, for asynchronous messages
    // handle the messages if any are found
    // The messages sorted out here should be removed from the queue.
}
```

`catch_response` : catches the response for the last command from the string queue. The below code snippet shows how to retrieve the response to the last command which is to get the UPS output status.

UPS is single-phase. ASCII protocol is used.

```
Response type:
  RESP_OUTPUT:SOURCE,VOLTAGE,FREQUENCY,CURRENT,POWER,LOAD:END
```

```
static int catch_response(int lastcmd)
{
    char *pstart = NULL;
    char *pend = NULL;
    int start, end, len;
    char respstr[1024] = {0, };

    switch(lastcmd)
    {
        .
        .
        .
        case OUTPUT:
            // search the sting buffer for the location of the response
            pstart = strstr(upsq.buf, "RESP_OUTPUT");
            pend = strstr(upsq.buf, "END");
            if(pstart && pend)
            {
                char *p;
                char *psource, *pvoltage, *pfrequency;
                char *pcurrent, *ppower, *pload;
                _UPS_DATA_OUTPUT output;
```

```
                    // read the current UPS output data
                    read_ups_output(&output);

                    // read the response
                    start = pstart - upsq.buf;
                    end = pend + strlen(END) - upsq.buf;
                    len = end-start;
                    queue_read(&upsq, respstr, start, len);

                    // interpret the response
                    p = respstr + strlen("REST_OUTPUT:");
                    if(p) psource = strtok(p, ",");
                    if(psource) pvoltage = strtok(NULL, ",");
                    if(pvoltage) pfrequency = strtok(NULL, ",");
                    if(pfrequency) pcurrent = strtok(NULL, ",");
                    if(pcurrent) ppower = strtok(NULL, ",");
                    if(ppower) pload = strtok(NULL, " ");

                    // save the data
                    output.num_lines = 1; // assume single-phase
                    if(psource) output.source = atoi(psource);
                    // Caution!! the units of frequency and current.
                    // See ups_data.h.
                    if(pfrequency) output.frequency = atoi(pfrequency) * 10;
                    if(pvoltage) output.voltage[0] = atoi(pvoltage);
                    if(pcurrent) output.current[0] = atoi(pcurrent) * 10;
                    if(ppower) output.power[0] = atoi(ppower);
                    if(pload) output.load[0] = atoi(pload);

                    // Save the output data to the UPS data area
                    save_ups_output(&output);

                    // Do the necessary jobs followed by the response
                    if(output.source == OUTSRC_NORMAL){
                         set_alarm(_ALARM_ONBATT, ALARM_RELEASED, true);
                         set_alarm(_ALARM_ONBYPASS, ALARM_RELEASED, true);
                    }
                    else if(output.source == OUTSRC_BATTERY){
                         set_alarm(_ALARM_ONBATT, ALARM_PRESENT, true);
                         set_alarm(_ALARM_ONBYPASS, ALARM_RELEASED, true);
                    }
                    else if(output.source == OUTSRC_BYPASS){
                         set_alarm(_ALARM_ONBATT, ALARM_RELEASED, true);
                         set_alarm(_ALARM_ONBYPASS, ALARM_PRESENT, true);
                    }
               }
               break;
          .
          .
          .
     }
     return 0;
}
```

`handle_upsmsg` : handles user commands from Web or SNMP server. The list of the `'command'` is defined in `ups_msgq.h`. The `'val'` and `'strval'` are the command specific arguments.

```
int handle_upsmsg(int command, long val, char *strval)
{
     // command string to UPS
     cmdstr[100] = {0, };

     switch(command)
```

```
    {
        // basic info
        case _UPSMSG_SET_ID_IDENTNAME: // set the name of a UPS
            sprintf(cmdstr, "SETIDNAME=%s", strval);
            break;
        case _UPSMSG_SET_ID_ATTDEVICE: // set the indentifier of
                                       //  the attached device to a UPS
            sprintf(cmdstr, "SETATTDEV=%s", strval);
            break;

        // battery
        case _UPSMSG_SET_BATT_REPLACEDATE:
            // date of the last battery replacement
            break;

        // UPS test
        case _UPSMSG_SET_TEST_ABORT:
            // test abort
            break;
        case _UPSMSG_SET_TEST_GENERAL:
            // general test
            break;
        case _UPSMSG_SET_TEST_QUICKBATT:
            // quick test
            break;
        case _UPSMSG_SET_TEST_DEEPBATT:
            // rigorous test
            break;
        case _UPSMSG_SET_TEST_USR:
            // user defined test, See chapter 4.
            break;

        // UPS contol
        case _UPSMSG_SET_CONT_SHUTDOWNTYPE:
            // set shutdown type, 'val' designates the type. (ups_data.h)
            break;
        case _UPSMSG_SET_CONT_SHUTDOWNAFDELAY:
            // schedule the shutdown, 'val' is set in seconds
            break;
        case _UPSMSG_SET_CONT_STARTUPAFDELAY:
            // schedule the startup, 'val' is set in seconds
            break;
        case _UPSMSG_SET_CONT_REBOOTWDURATION:
            // schedule the reboot, 'val' is set in seconds
            break;
        case _UPSMSG_SET_CONT_AUTORESTARTTYPE:
            // set the auto-restart type, 'val' designates
            //   the type (ups_data.h)
            break;

        // UPS configuration
        case _UPSMSG_SET_CONF_INPUTVOLT:
            // configure the input voltage, 'val' - [volts]
            break;
        case _UPSMSG_SET_CONF_INPUTFREQ:
            // configure the input frequency, 'val' - [Hz]
            break;
        case _UPSMSG_SET_CONF_OUTPUTVOLT:
            // configure the output voltage, 'val' - [volts]
            break;
        case _UPSMSG_SET_CONF_OUTPUTFREQ:
            // configure the output frequency, 'val' - [Hz]
            break;
        case _UPSMSG_SET_CONF_OUTPUTVA:
            // configure the output voltage/current rating,
            //   'val' - [volt-amp]
            break;
```

```
        case _UPSMSG_SET_CONF_OUTPUTPOWER:
            // configure the output power, 'val' - [watts]
            break;
        case _UPSMSG_SET_CONF_LOWBATTTIME:
            // configure the low battery time, 'val' - [min]
            break;
        case _UPSMSG_SET_CONF_AUDIBLESTATUS:
            // configure the audible alarm
            // 'val' -ADBLALM_STATUS in ups_data.h
            break;
        case _UPSMSG_SET_CONF_LOWVOLTTRPT:
            // configure the low battery transfer point
            // 'val' - [volts]
            break;
        case _UPSMSG_SET_CONF_HIGHVOLTTRPT:
            // configure the high battery transfer point
            // 'val' - [volts]
            break;
        case _UPSMSG_SET_CONF_BATTLIFE:
            break;

        // user defined control commands - See chapter 4.
        case _UPSMSG_SET_CONT_USR1:
        case _UPSMSG_SET_CONT_USR2:
        case _UPSMSG_SET_CONT_USR3:
        case _UPSMSG_SET_CONT_USR4:
        case _UPSMSG_SET_CONT_USR5:
        case _UPSMSG_SET_CONT_USR6:
        case _UPSMSG_SET_CONT_USR7:
        case _UPSMSG_SET_CONT_USR8:
        case _UPSMSG_SET_CONT_USR9:
        case _UPSMSG_SET_CONT_USR10:
            break;

        // user defined configuration commands - See chapter 4.
        case _UPSMSG_SET_CONF_USR1:
        case _UPSMSG_SET_CONF_USR2:
        case _UPSMSG_SET_CONF_USR3:
        case _UPSMSG_SET_CONF_USR4:
        case _UPSMSG_SET_CONF_USR5:
        case _UPSMSG_SET_CONF_USR6:
        case _UPSMSG_SET_CONF_USR7;
        case _UPSMSG_SET_CONF_USR8:
        case _UPSMSG_SET_CONF_USR9:
        case _UPSMSG_SET_CONF_USR10:
            break;
        default:
            break;
    }

    // send the string command to a UPS
    // or take specific action
    // The string commands differ with each UPS serial protocol.
    send_command(cmdstr);
}
```

`register_usrspec` : adds user defined data items. See chapter 4.

```
static void register_usrspec()
{
    // See chapter 4 to add codes here.
}
```

2.6. Modifying the Makefile and Compiling

Note : See Chapter 2.1 in advance.

If a user modified the `ups_app.c` and added `user_code.c` the current working directory will look like this.

```
#ls [enter]
Makefile    include    lib    ups_app.c    ups_mon.c    user_code.c
```

Change the `Makefile` to include the `user_code.c` in this project. The bold part of this `Makefile` is added for this purpose.

```
CROSS_COMPILE = /opt/hardhat/devkit/ppc/8xx/bin/ppc_8xx-
CC  = $(CROSS_COMPILE)gcc
AR  = $(CROSS_COMPILE)ar

INCLUDEDIRS = -I. -I./include
LDFLAGS = -L./lib
CFLAGS = ${INCLUDEDIRS} -D_REENTERANT

UPSAPP = upsapp
PROG_LIST = ${UPSAPP}

UPSAPP_OBJ = ./ups_mon.o ./ups_app.o ./user_code.o

all :  ${PROG_LIST}

${UPSAPP} : ${UPSAPP_OBJ}
	${CC} -o $@ ${LDFLAGS} ${UPSAPP_OBJ} -lupslink

c.o :
	${CC} -c ${CFLAGS} $<

clean :
	rm -f *.o ${UPSAPP_OBJ} ${PROG_LIST} core
```

Verify the output file, `upsapp`, after building the project.

```
#make clean; make[enter]
#ls [enter]
Makefile     include      lib            ups_app.c      ups_app.o
ups_mon.c    ups_mon.o    user_code.c    user_code.o    upsapp
```

If everything went OK, download the output into UPSLink. See the next chapter for this.

3. Download and Execution

3.1. Download

From the Web page of UPSLink download the UPS serial program.

Note : The file size of the serial program cannot exceed 128 Kbytes. A user cannot download the file from Telnet or Console connection. The downloading may take some time.

Location of this Web menu : UPS management ✍ UPS serial program

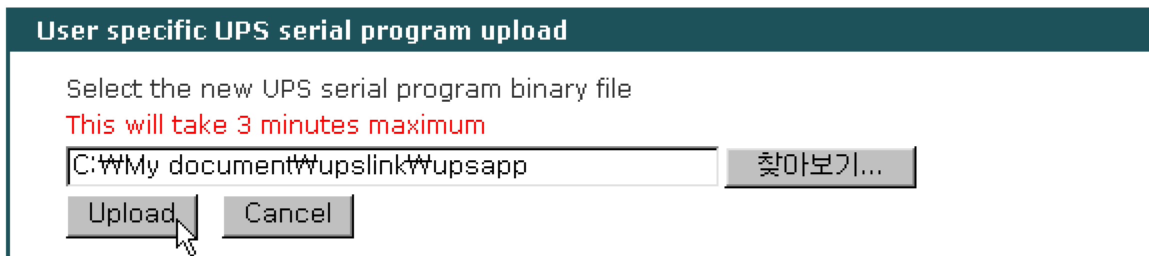

Figure 3- 1 Download a UPS serial program

A user can see a success message.

3.2. Running the UPS Serial Program

Once a new UPS serial program is downloaded into UPSLink, a user can run the program. Select 'User specific' from the combo box on the Web page and press the 'Apply' button.

Location of this Web menu : UPS management ✍ UPS serial program

Figure 3- 2 Running the UPS Serial Program

A new serial program starts.

4. Add User Defined Information

The user UPS serial program can accommodate user-defined data that are not defined in RFC1628.

The following items can be added.

- **UPS alarm** : Add an alarm item that is not described in RFC1628 but supported by a UPS. This item is defined with `upsAlarmDescr`.
- **UPS information** : A static information that is displayed on the Web. This is a static value.
- **UPS status information** : These values are read from UPS. For example, the number of the installed batteries.
- **UPS configuration** : These values are set to a UPS or read from it. Separate Web page is provided for these items.
- **UPS control** : UPS control items. Separate Web page is provided for these items.
- **UPS test** : Add a test item that is not described in RFC1628. This item is defined with `upsTestId`.

4.1. User Defined UPS Alarm

A user can add maximum 5 user defined UPS alarms, that are not defined in RFC1628.

Register a user defined alram

The `register_usralarm(..)` function is used to register a new alarm. This function is called from `register_usrspec()` in `ups_app.c`.

```
int register_usralarm(int usrindex, int oiddepth, unsigned long *oid,
char *shortdesc, char *longdesc);
```

`usrindex` : an index for an alarm, 0 ~ 4
`oiddepth` : the length of `oid`, maximum 20
`oid` : OID of `upsAlarmDescr` when an alarm is sent via SNMP trap
`shortdesc` : short description for an alarm, This is displayed when an alarm goes off. (maximum 31 bytes)
`longdesc` : long description for an alarm, (could be filed with blanks not with Null, maximum 255 bytes)

The `read_usralarm(..)` reads the registered list of user defined alarms.

```
int read_usralarm(int usrindex, struct _usr_alarm_object *alarmobj);
```

Example

Two user-defined alarms is to be added. The OIDs for the alarms are "1.3.6.1.4.1.12236.1" and "1.3.6.1.4.1.12236.2". The `'shortdesc's` are `"usr alarm 1"` and `"usr alarm 2"`. The below code snippet has to be added into `register_usrspec()`.

```
unsigned long oid1[8];
unsigned long oid2[8];
oid1[0] = 1; oid1[1] = 3; oid1[2] = 6; oid1[3] = 1; oid1[4] = 4; oid1[5] =
1; oid1[6] = 12236; oid1[7] = 1;
oid2[0] = 1; oid2[1] = 3; oid2[2] = 6; oid2[3] = 1; oid2[4] = 4; oid2[5] =
1; oid2[6] = 12236; oid2[7] = 2;
// The OIDs above has to be modified.
register_usralarm(0, 8, oid1, "usr alarm 1", "");
register_usralarm(1, 8, oid2, "usr alarm 2", "");
```

When an alarm with `usrindex = 0` has gone off,

```
set_alarm(_ALARM_USR1, ALARM_PRESENT, true);
```

When an alarm with `usrindex = 1` has been released,

```
set_alarm(_ALARM_USR2, ALARM_RELEASED, true);
```

Call the above functions to update alarm related UPS data when changed occur.

Note : Define `_ALARM_USR(1~5)` for `userindex`(0~4).

Note : See chapter 3 for `set_alarm()`.

Automatic alarm logging, SNMP trap and Email notification are followed as the system configuration. The status of user-defined alarms is displayed on the Web or Telnet.

The figure 4-1 shows that the user-defined alarm 1 is set.

Location of this Web menu : UPS management ✍ UPS status monitor

Figure 4- 1 User defined alarm is set

4.2. User Defined UPS Information

A user can add maximum 10 user defined UPS information that are not listed on the RFC1628.

Register user defined UPS information

The `register_usrinfo(..)` function is used to register a new UPS information. This function is called from `register_usrspec()` in `ups_app.c`.

```
int register_usrinfo(int usrindex, char *desc, INFO_TYPE type);
```

`usrindex` : an index for the new UPS information, 0~9
`desc` : description for the information, maximum 127bytes
`type` : type of the information, number or string (See `INFO_TYPE` in `ups_data.h`.)

The `read_ups_usrinfo(..)` and `save_ups_usrinfo`(..) reads and saves the user defined UPS

information.

```
int read_ups_usrinfo(int usrindex, int *infoval, char *infostr);
int save_ups_usrinfo(int usrindex, int infoval, char *infostr);
```

The `'infoval'` is numeric information and `'infostr'`(max. 63bytes) is string information.

Example

A user wants to show the width of a UPS and the contact point of an administrator on the UPSLink Web page.

Add these functions in the `register_usrspec()`.

```
register_usrinfo(0, "UPS physical dimension width [inch]", INFO_TYPE_NUM);
register_usrinfo(1, "UPS administrator's contact address", INFO_TYPE_TEXT);
```

These functions register the information and initialize as `0` or `NULL` characters.

If a user wants to change the information,

```
save_ups_usrinfo(0, 45, NULL);
save_ups_usrinfo(1, 0, "Utility room 104 (tel. 1234-5678)");
```

Figure 4-2 shows the screen of the relevant Web page.

Location of Web menu : UPS management ✍ UPS information

User specific information	
UPS physical dimension width [inch] :	45
UPS administrator's contact address :	Utility room 104 (tel. 1234-5678)

Figure 4- 2 User defined UPS information on the Web

4.3. User Defined UPS Status Information

A user can add maximum 10 user defined UPS status information that are not listed on RFC1628.

Register user defined UPS status information

The `register_usrstatus(..)` function is used to register a new UPS status information. This function is called from `register_usrspec()` in `ups_app.c`.

```
int register_usrstatus(int usrindex, char *desc, STATUS_TYPE type);
```

usrindex : an index for the information, 0~9
desc : description for the information, (maximum 127bytes)
type : type of the information. numeric or string, (See `STATUS_TYPE` in `ups_data.h`.)

The `read_ups_usrstatus(..)` and `save_ups_usrstatus(..)` reads and saves the user defined UPS status information.

```
int read_ups_usrstatus(int usrindex, int *statusval, char *statusstr);
int save_ups_usrstatus (int usrindex, int statusval, char *statusstr);
```

The `'infoval'` is numeric information and `'infostr'`(max. 63 bytes) is string information. (The

`'statusstr'` is maximum 63 bytes.)

Example

If a user wants to display the number of batteries to the UPS and the last startup time, add these functions in the `register_usrspec()`.

```
register_usrstatus(0, "Number of battery cells", STATUS_TYPE_NUM);
register_usrstatus(1, "Last boot up data and time", STATUS_TYPE_TEXT);
```

These functions register the information and initialize as `0` or `NULL` characters.

Note : A UPS is assumed to be smart to give information on these.

If a user gets the information as “20 batteries” and “2003—01-23-1723”, save them to the UPS data area by calling the following functions.

```
save_ups_usrstatus(0, 20, NULL);
save_ups_usrstatus(1, 0, "2003-01-23-1723");
```

The figure 4-3 shows the relevant Web page.

Location of the Web menu : UPS management ✍ UPS status monitor

User specific UPS status monitor
Number of battery cells : 20
Serial number of the UPS : MYUPS-0176583

Figure 4- 3 User defined UPS status information on the Web

4.4. User Defined UPS Configuration

A user can add maximum 10 UPS configuration items that are not listed on the RFC1628.

Once a new configuration is registered, a user can see a new Web page where he can send a configuration command to a UPS. The process of storing the new configuration value and sending the configuration command has to be implemented in `ups_app.c` by a user.

Note : The configuration value is written on the non-volatile area of memory.

See the following diagram.

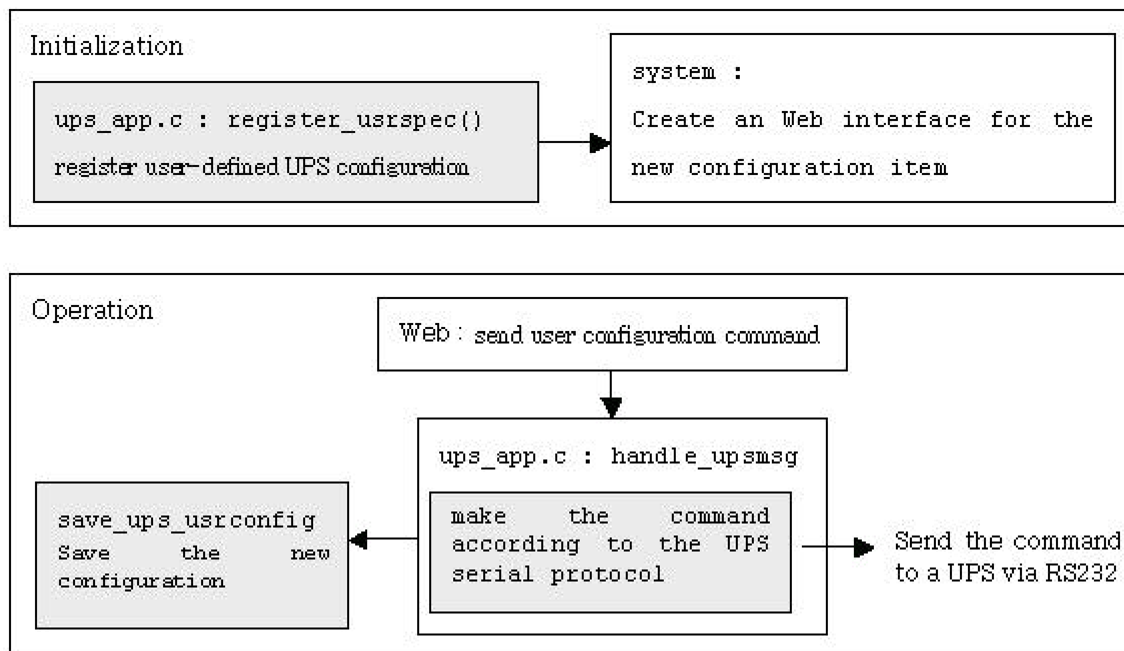

Figure 4- 4 User defined UPS configuration

The shaded box in the Figure 4-4 has to be modified or added by a user.

Register user defined UPS configuration

The `register_usrconfig(..)` function is used to register a new UPS configuration. This function is called from `register_usrspec()` in `ups_app.c`.

```
int register_usrconfig(int usrindex, char *desc, CONT_TYPE type, char **pldn_desc);
```

<u>`usrindex`</u> : an index for a new UPS configuration item, 0~9
<u>`desc :`</u> description for a new UPS configuration, (maximum 127 bytes)
<u>`type :`</u> type of a new UPS configuration, number, string or pull-down menu (See <u>`CONT_TYPE`</u> in <u>`ups_data.h`</u>.)
<u>`pldn_desc :`</u> used for the pull-down menu (maximum 10 menus up to 31 bytes long each)

The `read_ups_usrconfig(..)` and `save_ups_usrconfig(..)` reads and saves the user defined UPS status information.

```
int read_ups_usrconfig(int usrindex, int *configval, char *configstr);
int save_ups_usrconfig(int usrindex, int configval, char *configstr);
```

The `'configval'` is a numeric value or an index to the pull-down menu. The `'configstr'` is a string (maximum 63 bytes). It depends on the type of a configuration item.

Example

Add the boxed contents in the `register_usrspec()`. This will enable a user to configure maximum UPS output voltage, ID of the device attached to a UPS and the baudrate of the UPS serial

port.

```
char *pldn[] = {"1200", "2400", "4800", "9600", "19200", "38400",
     "57600", "115200", "230400", NULL};
register_usrconfig(0, "Max. output voltage [volt]", CONT_TYPE_NUM,  NULL);
register_usrconfig(1, "ID of attached device", CONT_TYPE_TEXT, NULL);
register_usrconfig(2, "Baudrate of the UPS [bps]", CONT_TYPE_PLDN, pldn);
```

Note : A UPS is assumed to be smart to be set for the configuration items on a serial command.

A user can view the figure below.

Location of the Web menu: UPS management ✍ UPS configuration

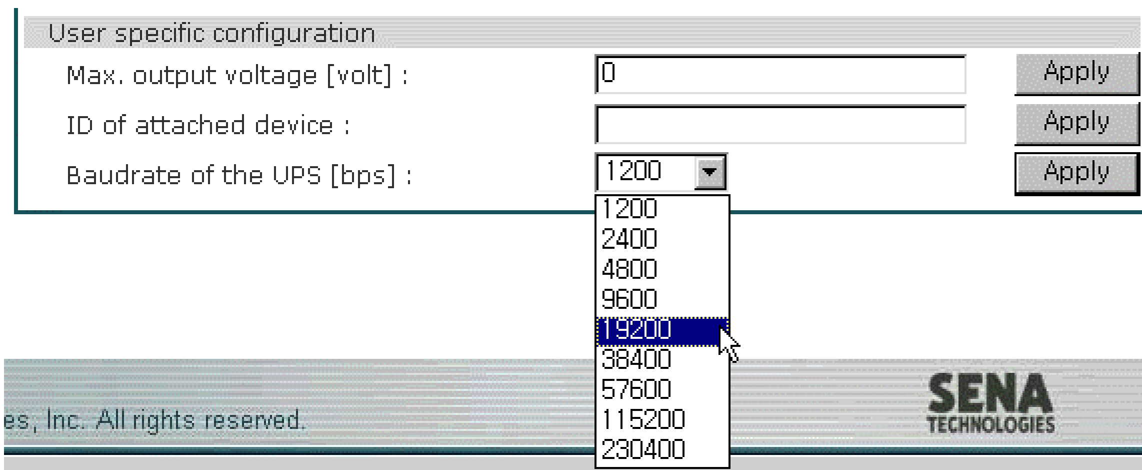

Figure 4- 5 Registered user defined UPS configuration

If a user reads 340(maximum output voltage), `ATTID` (ID of the attached device) and 9600 (baudrate of the UPS serial port) from a UPS, the following functions should be called in `ups_app.c`.

```
save_ups_usrconfig(0, 340, NULL);
save_ups_usrconfig(1, 0, "ATTID");
save_ups_usrconfig(2, 3, NULL);
```

The saved UPS configuration is viewed like the figure 4-6.

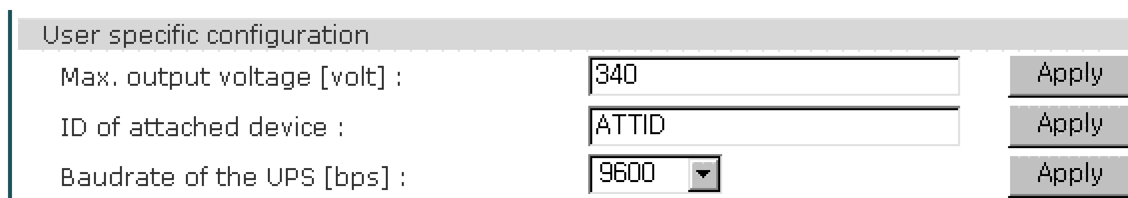

Figure 4- 6 Saved UPS configuration values

If a user press the 'Apply' button in Figure 4-5 the message delivery mechanism of UPSLink leads to the `handle_upsmsg(..)` in `ups_app.c`. The user should add a mechanism in the `handle_upsmsq(..)` to deliver the action on the Web page. Step into the code snippet for an example.

```
int handle_upsmsg(int command, long val, char *strval)
{
     char cmdstr[100] = {0, };
     switch(command)
```

```
    {
                .
                .
                .
        case UPSMSG_SET_CONF_USR1:
                sprintf(cmdstr, "SETMAXVOLT=%d", (int)val);
                // reconstruct the above command on a user's purpose
                break;
        case UPSMSG_SET_CONF_USR2:
                sprintf(cmdstr, "SETATTID=%s", strval);
                // reconstruct the above command on a user's purpose
                break;
        case UPSMSG_SET_CONF_USR3:
                {
                        char baudrate[10] = {0, };
                        switch(val)
                        {
                                case 0: strcpy(baudrate, "1200"); break;
                                case 1: strcpy(baudrate, "2400"); break;
                                case 2: strcpy(baudrate, "4800"); break;
                                case 3: strcpy(baudrate, "9600"); break;
                                case 4: strcpy(baudrate, "19200"); break;
                                case 5: strcpy(baudrate, "38400"); break;
                                case 6: strcpy(baudrate, "57600"); break;
                                case 7: strcpy(baudrate, "115200"); break;
                                case 8: strcpy(baudrate, "230400"); break;
                        }
                        sprintf(cmdstr, "SETBR=%s", baudrate);
                }
                // reconstruct the above command on a user's purpose
                break;
    }
    send_command(cmdstr); // send the command to UPS
}
```

Note : See chapter 3 for `handle_upsmsg(..)`.

If the configuration is of string type the `'strval'` argument is used to construct the serial command. Or if it's of numeric or pull-down menu type the `'val'` argument is used. (The `'val'` for the pull-down menu type of configuration item is an index to the menu.)

4.5. User Defined UPS Control

A user can add maximum 10 UPS control item that are not listed on the RFC1628.

Once a new control item is registered a user is given a new Web interface for that. The `'Apply'` button on this Web page delivers the relevant command to a UPS. A user should modify `ups_app.c` to implement this mechanism.

The user defined UPS control is mostly same as the user defined UPS configuration. Unlike the UPS configuration, nothing is saved for the UPS control items.

Refer to the diagram below.

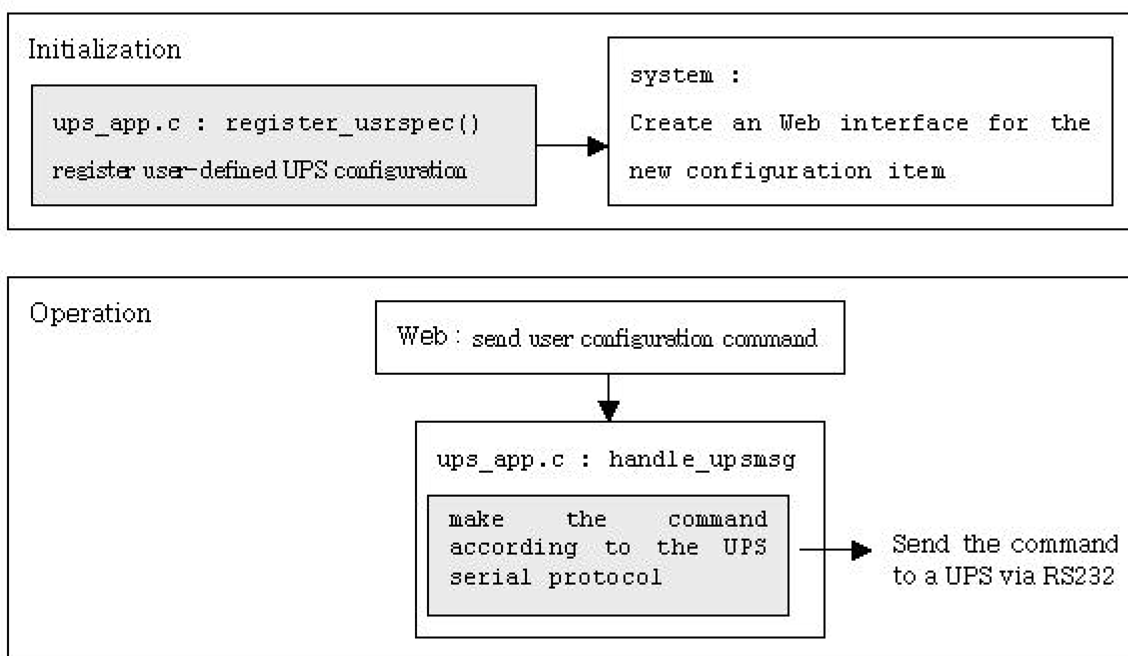

Figure 4- 7 User defined UPS control

The shaded box in the figure 4-7 has to be modified or added by a user.

Register user defined UPS contorl items

The `register_usrcontrol(..)` function is used to register a new UPS control item. This function is called from `register_usrspec()` in `ups_app.c.`

```
int register_usrcontrol(int usrindex, char *desc, CONT_TYPE type, int
defval, char *defstr, char **pldn_desc);
```

`usrindex` : an index to the UPS control item, 0~9
`desc` : description for the UPS control item. (maximum 127 bytes)
`type` : type of the UPS control item, number, string or pull-down menu (See `CONT_TYPE` in `ups_data.h`.)
`defval` : default value of a numeric or pull-down type of a UPS control item
`defstr` : default string of a string type of a UPS control item
`pldn_desc` : pull-down menu items (maximum 10 menus up to 31 bytes long each)

Control items do not save any data in the UPSLink data area. So the `'save'` and `'read'` functions are not provided. The `'defval'` or `'defstr'` are always displayed on the Web.

Example

To register a user defined UPS control item is the same as a user defined UPS configuration except a registering function. When modifying the `ups_app.c`, `UPSMSG_SET_CONT_USR1~10` is used instead of `UPSMSG_SET_CONF_USR1~10` in `handle_upsmsg(..)`.

Note : See the example in chapter 4.5.

4.6. User Defined UPS Test

A user can add maximum 5 UPS tests that are not listed on the RFC1628. Each new UPS test is assigned a specific `upsTestId`.

Register user defined UPS test

The `register_usrtest(..)` function is used to register a new UPS control item. This function is called from `register_usrspec()` in `ups_app.c`.

```
int register_usrtest(int usrindex, int oiddepth, unsigned long *oid,
char *testdesc);
```

`usrindex` : index to a new user defined UPS test. 0~4.
`oiddepth` : length of OID. maximum 20.
`oid` : OID of upsTestId. This value is an answer to SNMP-get.
`testdesc` : description for a new test. (maximum 63 bytes)

Example

The boxed contents are added in the `register_usrspec()` to post a new UPS rectifier test and inverter test on the Web.

```
unsigned long oid1[8];
unsigned long oid2[8];
oid1[0] = 1; oid1[1] = 3; oid1[2] = 6; oid1[3] = 1; oid1[4] = 4; oid1[5] =
1; oid1[6] = 12236; oid1[7] = 1;
oid2[0] = 1; oid2[1] = 3; oid2[2] = 6; oid2[3] = 1; oid2[4] = 4; oid2[5] =
1; oid2[6] = 12236; oid2[7] = 2;
// The two OIDs above has to be modified for a user's purpose.
register_usrtest(0, 8, oid1, "Rectifier test");
register_usrtest(1, 8, oid2, "Inverter test");
```

Note : If the tests are registered as above the UPS rectifier test could be on the Web and the OID of the last test, .1.3.6.1.4.1.12236.1, is returned for SNMP-Get query for `upsTestID`.

Note : A UPS is assumed to be smart to do this job.

If a new UPS test is registered the test description is added on the pull-down menu like figure 4-8.

Location of the Web menu : UPS management ✍ UPS control

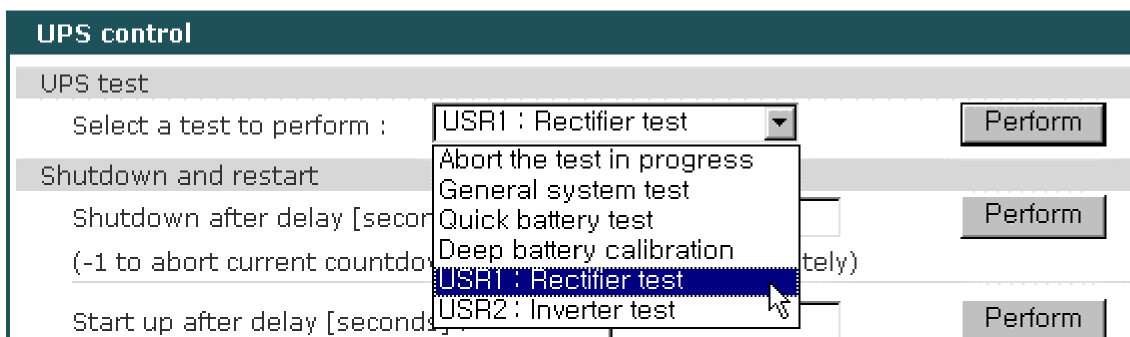

Figure 4- 8 Registered user defined UPS test

If a user pushes the `'Perform'` button on the Web the `handle_upsmsg(..)` in `ups_app.c` is called by underlying message delivery mechanism. To actually send a command for this user action the `handle_upsmsg(..)` has to be modified. See the example below.

```
int handle_upsmsg(int command, long val, char *strval)
{
     char cmdstr[100] = {0, };
     switch(command)
     {
               .
               .
               .

          case _UPSMSG_SET_TEST_ABORT:
               save_ups_testid(TEST_ABORT, 0);
               sprintf(cmdstr, "TESTABORT");
               // reconstruct the command for a user specific protocol
               break;
          case _UPSMSG_SET_TEST_GENERAL:
               save_ups_testid(TEST_GENERAL, 0);
               sprintf(cmdstr, "GENERALTEST");
               // reconstruct the command for a user specific protocol
               break;
          case _UPSMSG_SET_TEST_QUICKBATT:
               save_ups_testid(TEST_QUICKBATT, 0);
               sprintf(cmdstr, "QUICKBATTTEST");
               // reconstruct the command for a user specific protocol
               break;
          case _UPSMSG_SET_TEST_DEEPBATT:
               save_ups_testid(TEST_DEEPBATTCALIB, 0);
               sprintf(cmdstr, "DEEPBATTTEST");
               // reconstruct the command for a user specific protocol
               break;
          case _UPSMSG_SET_TEST_USR:
               switch(val)
               {
                    case 0:
                         save_ups_testid(TEST_USER_SPECIFIC, 0);
                         sprintf(cmdstr, "RECTIFIERTEST");
                         break;
                    case 1:
                         save_ups_testid(TEST_USER_SPECIFIC, 1);
                         sprintf(cmdstr, "INVERTERTEST");
                         break;
               }
               // reconstruct the command for a user specific protocol
               break;
     }
     send_command(cmdstr); // send the test command to UPS
}
```

`_UPSMSG_SET_TEST_USR` is passed to the `'command'` argument for all the user defined UPS tests. Each test is identified by the `'val'` argument that is an index to the test.

Note : Call `save_ups_testid`(..) after sending a test command to a UPS. This saves the `upsTestId` and the test start-time.

Note : See chapter 3 for `handle_upsmsg(..)` and `save_ups_testid(..)`.